

LinuxFocus article number 273
http://linuxfocus.org

Securiser ses connexions avec SSH



par Bernard Perrot
<bernard.perrot(at)univ-rennes1.fr>

L´auteur:

Bernard est Ingénieur Système et Réseau au CNRS depuis 1982. Il a par exemple, été chargé de mission à la sécurité informatique à l'Institut National de Physique Nucléaire et de Physique des Particules (In2p3). Il travaille maintenant dans un institut de recherches en mathématiques (IRMAR) dans le département SPM (Sciences Physique et Mathématiques).



Résumé:

Cet article a été publié dans un numéro spécial sur la sécurité de Linux Magazine France. L'éditeur, les auteurs, les traducteurs ont aimablement accepté que tous les articles de ce numéro hors-série soient publiés dans LinuxFocus. En conséquence, LinuxFocus vous "offrira" ces articles au fur et à mesure de leur traduction en Anglais. Merci à toutes les personnes qui se sont investies dans ce travail. Ce résumé sera reproduit pour chaque article ayant la même origine.

Cet article a pour ambition de vous faire connaître SSH, à quoi il sert, et vous donner les éléments de choix pour décider de l'utiliser ou non (sauf que si vous décidez de ne pas l'utiliser, j'ai raté mon coup...). Ce n'est pas un mode d'emploi didactique, ni un manuel d'installation, mais plutôt une introduction au vocabulaire et fonctionnalités de SSH. Les liens et documentations indiquées ici contiennent en détail toutes les informations nécessaires au déploiement, en mieux que ce que je pourrais (mal) résumer dans les quelques paragraphes qui suivent.

A quoi sert SSH ?

Au premier ordre (et historiquement), SSH (la commande *ssh*) est une version sécurisée de *rsh* (et *rlogin*). SSH veut dire "*Secure SHell*" à l'image de *rsh* qui veut dire "*Remote SHell*". Donc, quand *rsh* permet d'obtenir un shell distant aisément, mais sans mécanisme d'authentification satisfaisant (du point de vue de la sécurité), *ssh* procure le même service de façon (hautement) sécurisée (en cumulant plusieurs niveaux pour assurer la sécurité).

Si on voulait faire très bref pour l'utilisateur basique qui ne veut pas en savoir plus (ni en faire plus), on pourrait s'arrêter ici en affirmant (à raison) que une fois que l'administrateur des machines serveurs et

clientes a correctement effectué son travail (cela est relativement aisé désormais), il suffit d'utiliser la commande *ssh* à la place des commande *telnet*, *rsh* et *rlogin*, et tout marche pareil, mais en plus sécurisé (donc avec une bénéfice au pire égal, en général très supérieur et cela à très moindre effort).

Donc, quand vous faisiez :

`% rlogin serveur.org` (ou `telnet serveur.org`)

vous faites :

`% ssh serveur.org`

et c'est déjà bien mieux !

Et pour terminer dans le bref, je m'avancerais à dire que aujourd'hui, tout incident de sécurité qui aurait pu être contenu par le seul et simple usage de SSH à la place de *rsh* (*rlogin*, *telnet*) relève plus de la négligence des parties victimes que de l'absolue fatalité (pour tempérer, on admettra qu'il n'y aura négligence qu'à partir du moment ou il n'y avait pas ignorance de la solution).

Quels sont les besoins ?

Pour détailler plus, voici quelques-uns des aspects sensibles et fragiles des connexions interactives que l'on souhaiterait voir résolus :

- en premier lieu, éviter la compromission des mots de passe, qui circulent "*en clair*" sur le réseau;
- disposer d'une authentification renforcée des machines, pas seulement basée sur le nom ou l'adresse IP, fortement sensibles à la mascarade;
- pouvoir exécuter en toute sécurité des commandes à distance;
- pouvoir transférer des fichiers en toute sécurité;
- sécuriser les sessions X11, très vulnérables;

A ces besoins, il existe des solutions connues, non satisfaisantes :

- les "*R-commandes*" : elles évitent en effet la circulation des mots de passe en clair, mais en ayant recours au mécanisme des "rhosts" qui posent alors de très gros problèmes de sécurité;
- les "*mots de passe à usage unique*" (*One Time Password*, OTP) : ce mécanisme ne protège que l'authentification (pas la communication ensuite), et bien que ce soit un système très séduisant d'un point de vue sécurité, il est très contraignant, difficile à faire accepter, et de ce fait se trouve réservé à des situations / contextes bien particuliers (dans lesquels la contrainte sur l'ergonomie est acceptable);
- telnet avec chiffrement : cette solution ne couvre ... que telnet. En particulier, cela ne fournit pas de réponse pour le protocole X11, indispensable pour beaucoup en complément.

Et il existe SSH, qui va :

- remplacer les *R-commandes* : *ssh* va se substituer à *rsh* et *rlogin*, *scp* à *rcp*, *sftp* à *ftp*;

- procurer une authentification forte, basée sur l’algorithmique cryptographique à clés publiques (aussi bien des machines que des utilisateurs);
- permettre de rediriger tout flux TCP dans le "tunnel" de la session, et en particulier X11 qui peut l’être automatiquement;
- chiffrer le tunnel, et au besoin et sur demande, le compresser.

SSH version 1 et SSH version 2

Comme rien n’est parfait en ce monde, il y a deux versions, incompatibles de SSH (du protocole) : les versions 1.x (1.3 et 1.5), et la version 2.0. Le passage d’une version à l’autre est sans douleur pour l’utilisateur, au détail près qu’il faut avoir le bon client avec le bon serveur pour conséquence de cette incompatibilité.

Le protocole est monolithique dans SSH version 1, tandis que SSH version 2 a redéfini celui-ci en trois "couches" :

1. SSH Transport Layer Protocol (SSH-TRANS)
2. SSH Authentication Protocol (SSH-AUTH)
3. SSH Connection Protocol (SSH-CONN)

chacun faisant l’objet d’un document de spécification particulier (normalisé à l’IETF), accompagné d’un quatrième décrivant l’architecture (SSH Protocol Architecture, SSH-ARCH). On trouvera tous les détails à l’adresse : http://www.ietf.org/html.charters/secsh-charter.html

Sans entrer trop dans les détails, indiquons cependant que dans SSHv2:

- la couche transport prend en charge l’intégrité, le chiffrement et la compression, l’authentification des machines
- la couche authentification prend en charge ... l’authentification (password, hostbased, clés publiques)
- la couche connexion prend en charge la gestion du tunnel (shell, agent SSH, redirection de ports, contrôle de flux).

Les différences techniques essentielles entre les versions 1 et 2 sont :

SSH version 1	SSH version 2
conception monolithique	séparation des couches authentification, connexion et transport
intégrité via CRC32 (peu fiable)	intégrité via HMAC (hash cryptographique)
un et un seul canal par session	nombre indéterminé de canaux par session
négociation du seul chiffrement symétrique du tunnel, clé de session unique pour les deux sens	négociations plus détaillées (chiffrement symétrique, clés-publiques, compression, ...), et clés de session, compression et intégrité séparées pour les deux sens
RSA seulement pour l'algorithmique clés-publiques	RSA et DSA pour l'algorithmique clés-publiques
clé de session transmise par le client	clés de session négociées avec un protocole Diffie-Hellman
clé de session valable pour toute la session	clés de session renouvelées

Le trousseau de clés

SSH utilise plusieurs types de clés, voici leurs définitions rapides :

- la "clé utilisateur" ("*user key*") : c'est une paire clé publique/clé privée (un bi-clé asymétrique), créée par l'utilisateur, et permanente (stockée sur disque). Elle permet l'authentification de l'utilisateur si ce mode d'authentification à clé publique est utilisé (voir ci-après)
- la "clé hôte" ("*host key*") : c'est une paire clé publique/clé privée (un bi-clé asymétrique), créée par l'administrateur du serveur, en général là l'installation/configuration du produit. Elle est permanente et stockée sur disque. Elle permet l'authentification des machines entre-elles.
- la "clé serveur" ("*server key*") : c'est une paire clé publique/clé privée (un bi-clé asymétrique), générée par le démon au démarrage et régulièrement régénérée. Elle reste uniquement en mémoire, et est utilisée dans SSHv1 pour sécuriser l'échange de la clé de session (en SSHv2, cet échange utilise un protocole Diffie-Hellman et cette clé n'existe plus).
- la "clé de session" ("*session key*") : c'est une clé secrète, destinée à être utilisée par l'algorithme de chiffrement symétrique chiffrant le canal de communication. Elle est aléatoire et volatile (comme toujours dans les produits de cryptographie modernes, l'algorithmique à clé publique sert à assurer le secret de l'échange des clés secrètes aléatoires et volatiles). En SSHv1, il y en a une par session, identique pour les deux sens de communication. En SSHv2, il y en a une par sens de communication (donc deux par session, régénérées).

A cela s'ajoute la "*passphrase*" de l'utilisateur, qui permet de protéger la composante privée du bi-clé asymétrique de celui-ci, cette protection étant assurée en chiffrant à l'aide d'un algorithme symétrique le fichier contenant la clé privée. La clé secrète utilisée pour chiffrer ce fichier est dérivée de cette "*passphrase*".

Les méthodes d’authentification

Il existe plusieurs méthodes d’identification des utilisateurs, à choisir en fonction des besoins, de la politique de sécurité. Le fichier de configuration du serveur permet d’autoriser ou non telle et telle méthode en fonction de cette politique. Voici les principales catégories :

- **"à la telnet":**

  Il s’agit de l’authentification "traditionnelle" par mot de passe : lors de la connexion, après avoir décliné son identité, l’utilisateur est invité à entrer un mot de passe qui est transmis au serveur, qui le compare à celui associé à l’utilisateur (je simplifie, en général il est comparé à une empreinte de ce password afin de ne pas stocker celui-ci en clair sur un support lisible). Le problème (celui encore cause d’un nombre sidéral de piratages sur Internet) est que ce mot de passe circule *en clair* sur le réseau, et donc peut être intercepté par n’importe qui à l’aide d’un simple "*sniffer*". L’apport de SSH ici est que tout va fonctionner à l’identique (il s’agit donc de la méthode facile à apprendre pour un débutant migrant de *telnet* à SSH puisqu’il n’y a rien à apprendre...), sauf que le canal étant déjà chiffré par le protocole, le password *clair* est encapsulé dans une communication secrète, et devient donc inviolable (sur le réseau).

  Une variante plus sécurisé encore, configurable si on possède ce qu’il faut sur le poste serveur est l’utilisation de "*mots de passe à usage unique*" (S/Key en l’occurrence) : c’est sans doute très bien, encore plus sûr évidemment, mais l’ergonomie contraignante de ce système le réserve sans doute à des sites bien particuliers. Rappelons que le principe des systèmes de "*mots de passe à usage unique*" est le suivant : après avoir décliné son identité, au lieu de demander son mot de passe (statique) à l’utilisateur, le serveur lui soumet un "*challenge*", auquel il doit répondre. Ce challenge étant toujours différent, la réponse le sera également. Ainsi donc, le fait d’intercepter cette réponse est sans importance puisque cela ne peut pas resservir. La contrainte indiquée vient du fait essentiel que cette réponse nécessite un dispositif pour être calculée (calculette externe, utilitaire logiciel sur la machine cliente, etc.), et que l’entrée de la réponse est très "cabalistique" (six monosyllabes anglo-saxonnes dans le meilleur des cas).

- **"à la rhosts"**" (*hostbased*) :

  Il s’agit d’une identification similaire à celle pratiquée avec les R-commandes et les fichiers tels que /etc/rhosts ou ~/.rhosts, qui "certifient" les sites clients. Le seul apport de SSH est l’identification accentuée des sites, et l’utilisation de fichiers "*shosts*" privatifs, mais c’est peu et je déconseille cette méthode utilisée seule.

- **Par clés publiques**

  Ici, l’authentification sera réalisée par un système basé sur la cryptographie asymétrique (voir cet article; SSH version 1 utilise RSA pour ce mode, et SSH version 2 a introduit DSA.), la clé publique de l’utilisateur étant (préalablement) déposée sur la machine serveur, et sa clé privée stockée sur sa machine cliente. Avec un tel système d’authentification, aucun secret ne circule sur le réseau et n’est jamais fourni au serveur.

  Ce système est excellent, à la contrainte près (à mon avis) que sa sécurité repose quasi exclusivement sur le "sérieux" de l’utilisateur (ce problème n’est alors pas particulier à SSH, mais

est à mon sens LE problème majeur des systèmes à clés publiques, tels que les PKIs actuellement à la mode) : en effet, afin d'éviter la compromission de la clé privée sur la machine cliente, celle-ci est normalement elle-même protégée par mot de passe (on parle ici en général de *passphrase* afin de notifier qu'il n'est pas raisonnable de s'en tenir à un seul *mot*). Si l'utilisateur ne protège pas correctement (pas du tout...) sa clé privée, celle-ci peut être utilisée par un tiers, qui aura alors accès à toutes les ressources de l'utilisateur. Je dis que cette sécurité ne repose que sur le sérieux, la confiance de l'utilisateur, car dans un tel système, l'administrateur de la machine serveur n'a *aucun* moyen de savoir si cette clé privée est protégée ou non, et actuellement, SSH ne sait pas gérer de listes de révocation (pas grand monde ne les gèrent d'ailleurs non plus dans les PKIs actuellement...). Par exemple, si une clé privée est stockée sans passphrase sur une machine domestique à domicile (il n'y a pas de méchants à domicile, à quoi bon s'embêter avec une passphrase...?), et que la dite machine part en dépannage au SAV d'une la grande surface quelconque (ne riez pas, c'est ce qui va arriver quand la signature électronique va se développer...), le dépanneur (son fils, ses copains) pourra récupérer les clés privées de tout ce qui va passer sur sa table.

La configuration de ce mécanisme d'authentification pour l'utilisateur est légèrement différente selon que l'on utilise SSHv1, SSHv2 ou OpenSSH, ou bien un client MacOS^tm ou Windows^tm. Les principes et étapes à retenir sont :

- génération d'un "bi-clé asymétrique" (c'est à dire un couple clé-privée/clé-publique RSA ou DSA) sur une machine, en général la machine cliente (si on utilise plusieurs machines clientes, on effectue cette génération en général sur une seule d'entre elles, puis on recopie les clés sur les autres). Certains clients Windows^tm et MacOS^tm n'ont pas d'utilitaire de génération des bi-clés, il faut alors les générer sur une machine Unix, puis les recopier après sur la machine voulue. Ce bi-clé est stocké dans un sous-répertoire `~/.ssh` de l'utilisateur.
- copie de la clé publique sur les machines serveur sur lesquelles on souhaite pouvoir utiliser cette authentification. Cela consiste à ajouter la ligne correspondant à la clé publique présente dans le répertoire où elle a été générée dans un fichier du serveur, dans le même répertoire `~/.ssh` de l'utilisateur (le nom dépend de la version de SSH, `authorized_keys` ou `authorization`).
- et c'est tout... si ce type d'authentification a été configuré au niveau du serveur, le client demandera alors la "*passphrase*" au moment de le séquence de connexion.

En complément, il est utile de connaître au moins les deux points suivants à propos de l'authentification :

- **ssh-agent**

  Une des raisons qui incite à ne pas protéger sa clé privée est l'agacement à devoir entrer celle-ci à chaque utilisation, et de plus l'impossibilité de le faire si on souhaite utiliser SSH dans des scripts détachés. A cela, il existe une réponse, l'*agent* SSH : c'est un utilitaire (*ssh-agent*) qui, une fois activé sous votre contrôle, va vous permettre de stocker des triplés identifiants (username/hostname/passphrase), et les soumettra à votre place lorsqu'ils seront requis par une connexion. Comme cela demande de ne donner qu'une seule fois son mot de passe par connexion sur le poste client, on peut dire que c'est une sorte de SSO (*Single Sign On*).

  Voilà, vous êtes prévenus, rien ne vous contraint à ne pas protéger vos clés privées, si vous le

faites, ce sera de la négligence, vous serez responsable des conséquences.

- **mode "*verbose*"**

  Il arrive que la connexion échoue pour une raison qui échappe à l'utilisateur : utiliser alors l'option "`-v`" (pour "*verbose*") de la commande *ssh*. Celle-ci va avoir pour conséquence de détailler par de très nombreux messages affichés à l'écran la séquence de connexion, et très souvent permettre de découvrir la cause du rejet.

Les algorithmes de chiffrement

Il faut distinguer ceux servant au chiffrement du canal de communication (chiffrement à clés secrètes) et ceux servant à l'authentification (chiffrement à clés publiques).

Pour l'authentification, on a le choix entre RSA et DSA avec la version 2 du protocole, et il n'y a que RSA pour la version 1 (donc pas de choix....). Le choix de DSA s'imposait historiquement si RSA n'était pas utilisable pour cause de *patent* sur le produit dans certains pays. Depuis la fin de l'été 2000, RSA est libre de droits, et donc cette contrainte a disparu. Je n'ai pas d'avis précis sur le bon ou mauvais choix (signalons quand même que DSA est un "pur produit" de la NSA, mais qu'est-ce qui ne l'est pas directement ou indirectement actuellement dans la cryptographie grand public et commerciale...?).

Pour le chiffrement symétrique, il y a presque trop le choix... Le protocole impose un algorithme commun et devant être présent dans toutes les implémentations : le *triple-DES à trois clés*. Par conséquent, c'est celui qui sera utilisé si la négociation entre le client et le serveur échoue sur une autre concordance. Cependant, s'il est possible d'en négocier un autre, c'est préférable, le 3DES étant maintenant un des moins performants. On écartera sauf nécessité les exotiques ou trop anciens (arc4, DES, RC4, ...) pour se limiter à :

- IDEA : plus performant que le 3DES, mais non totalement libre de droits dans certaines conditions (c'était souvent celui configuré par défaut dans la version Unix);
- Blowfish : très rapide, probablement sûr, mais algorithme en fait peu éprouvé;
- AES : le nouveau standard (le remplaçant du DES), s'il est disponible pour les deux parties, le prendre, il est fait pour ça.

Personnellement, je m'interroge sur l'utilité de proposer autant d'algorithmes : que le protocole prévoit la possibilité d'en négocier un "privatif" (pour un cercle d'usagers particulier par exemple), cela me semble indispensable, mais pour l'usage courant, je pense qu'à terme, AES a vocation à devenir le standard, et alors pourquoi ne pas s'en tenir à lui. Si AES devait être compromis, les soucis de sécurité dépasseraient très largement ceux induits dans SSH...

Redirection de ports, tunneling

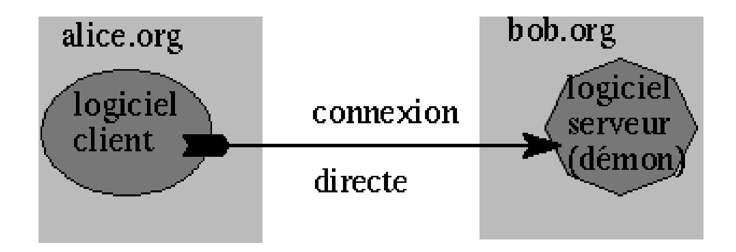
SSH permet de rediriger (*forward*) n'importe quel flux TCP dans le "tunnel" de la session SSH. Cela

veut dire que le flux de l'application considérée, au lieu d'être véhiculé entre les ports client et serveur habituels directement, le sera "encapsulé" à l'intérieur du "tunnel" créé par la connexion (session) SSH (*voir shéma ci-dessous*).

Il le fait sans effort (pour l'utilisateur) pour le protocole X11, avec gestion transparente des *displays* et propagation en continuité de ceux-ci lors de connexions en cascades.

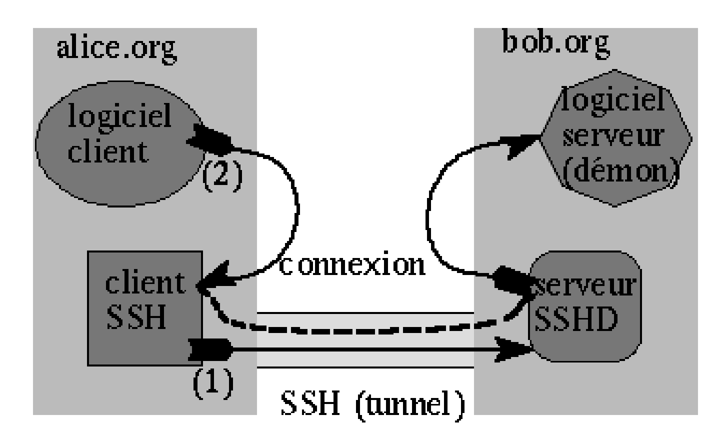
Il le fait à la demande pour d'autres flux, grâce à une option de la ligne de commande, et ce dans les deux sens au choix :

- Connexion directe entre client et serveur

  (exemple : `user@alice% telnet bob.org`)

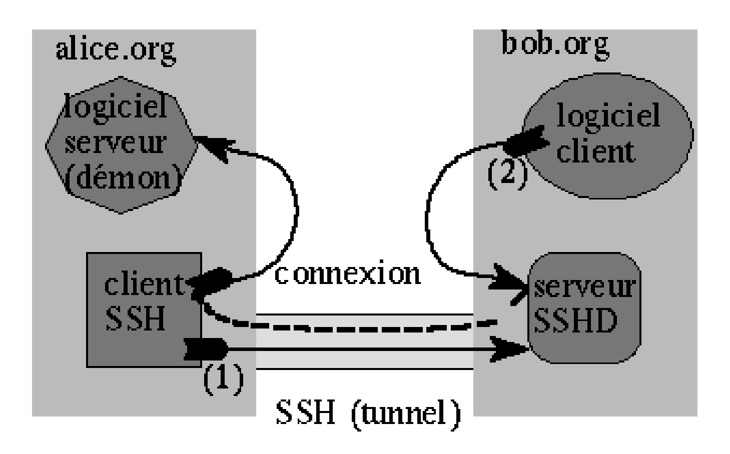
- Redirection de port local (client) vers port distant (serveur)

  exemple : `user@alice% ssh -L 1234:bob.org:143 bob.org`

  ce système permet d'accéder depuis "*alice.org*" au serveur imap de "*bob.org*", qui refuserait les connexions externes à son réseau local, et sera alors accessible à l'adresse *localhost*, port 1234, vu du client imap exécuté sur "*alice.org*".

  - (1) l'utilisateur sur "*alice.org*" ouvre (connexion) le tunnel SSH
  - (2) l'utilisateur sur *"alice.org"* configure le client local imap pour qu'il accède au serveur imap *localhost* sur le port 1234
- Redirection de port distant (client) vers port local (serveur)



exemple : `root@alice% ssh -R 1234:bob.org:143 bob.org`

ce système permet d'accéder depuis "*bob.org*" au serveur imap de "*alice.org*", qui refuserait les connexions externes à son réseau local, et sera alors accessible à l'adresse *localhost*, port 1234, vu du client imap exécuté sur "*bob.org*".
Dans le cas présent, l'initiative d'ouvrir le tunnel est prise par l'administrateur de "*alice.org*" (qui possède donc les privilèges requis pour exécuter cette commande ssh) qui désire offrir une solution pour accèder cependant à son serveur imap.

- (1) l'administrateur sur "*alice.org*" ouvre (connexion) le tunnel SSH
- (2) l'utilisateur sur *"bob.org*" configure le client local imap pour qu'il accède au serveur imap *localhost* sur le port 1234

Cette fonctionnalité très puissante vaut parfois à SSH l'appellation de "tunnel du pauvre". Il faut comprendre ici que la notion de pauvreté est celle qui signifie ne pas posséder de privilèges (droits) administrateur sur la machine cliente, car, sauf cas particuliers, la redirection d'un port local (non privilégié, donc supérieur à 1024) ne nécessite pas de droits super-utilisateur ("*root*") pour être utilisée. Par contre, la redirection d'un port local privilégié devra être réalisée par un compte *root*, ou bien le client devra être installé avec les droits du super-utilisateur ("*suid*") (car la redirection d'un port local privilégié permet en fait de redéfinir un service standard).

Comme avec IP, il est assez aisé de mettre tout dans tout (et réciproquement), il est non seulement possible de rediriger ainsi des flux TCP particuliers, mais également un flux PPP, ce qui permet de réaliser un "vrai" tunnel IP dans IP (qui plus est chiffré, donc sécurisé). La description complète de la méthode dépasse le cadre de ce court article, on pourra se référer au "Linux VPN-HOWTO" pour détails et script de mise en oeuvre (il existe également des solutions natives de VPN sous Linux, comme "*stunnel*" qu'il faut également considérer avant de faire un choix définitif).

A noter qu'une des premières possibilités de la redirection est de rediriger le flux *telnet* : cela peut sembler totalement inutile, puisque SSH implémente la connexion interactive par défaut. Cependant, en redirigeant le flux *telnet*, il est possible d'utiliser un client préféré à la place du mode interactif natif de SSH. Cela peut être peut être particulièrement apprécié en environnement Windows^{tm} ou MacOS^{tm} où le client SSH peut ne pas posséder l'ergonomie favorite de l'utilisateur. Par exemple, la partie "émulation de terminal" du client "*Mindterm*" (client SSH en Java, donc fonctionnel sur tout système moderne) souffre du manque de performance du langage : il peut être avantageux d'utiliser ce client

uniquement pour ouvrir le tunnel SSH, puis d’y faire passer y compris son client *telnet* favori.

On peut également par ce biais lancer un client du type "*xterm*" distant (en l’occurrence à travers la redirection X11 automatique de SSH), ce qui permet l’utilisation de SSH sur des dispositifs ne possédant pas de client natif (terminaux X par exemple, cependant, il faut une machine délégataire ("*proxy*") sur le réseau local et la connexion entre ce *proxy* et le TX n’est pas sécurisée).

Notez que le tunnel est maintenu ouvert tant qu’il y a un flux redirigé qui l’utilise, même s’il ne s’agit pas de celui qui l’a créé. Ainsi, la commande "*sleep*" devient très utile pour ouvrir un tunnel SSH et y rediriger un autre flux TCP :

```
% ssh -n -f -L 2323:serveur.org:23 serveur.org sleep 60

% telnet localhost 2323

... welcome to serveur.org ...
```

La première ligne ouvre le tunnel, exécute la commande "*sleep 60*" sur le serveur, et redirige le port local 2323 vers le port distant 23 (*telnet*). La deuxième lance un client *telnet* sur le port local 2323, et donc va utiliser ce tunnel (chiffré) pour se connecter au démon *telnetd* du serveur. Le "*sleep*" va s’interrompre après une minute (il n’y a donc qu’une minute pour lancer le telnet), mais SSH ne fermera le tunnel que lorsque la connexion du dernier client l’utilisant sera terminée.

Principales distributions gratuites disponibles

Il va falloir distinguer la disponibilité des clients et/ou serveurs selon les plates-formes, et pour cause d’incompatibilité, entre version 1 et version 2. Les références en fin d’article vont permettront de trouver les implémentations non-citées ici, ce tableau se cantonne à des produits gratuits et suffisamment stables et fonctionnels.

produit	plateforme	protocole	lien	notes
OpenSSH	Unix	versions 1 et 2	www.openssh.com	voir ci-dessous
TTSSH	Windows^{tm}	version 1	www.zip.com.au/~roca/ttssh.html	
Putty	Windows^{tm}	version 1 et 2	www.chiark.greenend.org.uk/~sgtatham/putty	bêta seulement
Tealnet	Windows^{tm}	version 1 et 2	telneat.lipetsk.ru	
SSH secure shell	Windows^{tm}	versions 1 et 2	www.ssh.com	gratuit pour usage non commercial
NiftytelnetSSH	MacOS^{tm}	version 1	www.lysator.liu.se/~jonasw/freeware/niftyssh/	
MacSSH	MacOS^{tm}	version 2	www.macssh.com	
MindTerm	Java	version 1	www.mindbright.se	versions 2 commerciales

A noter que MindTerm est à la fois une implémentation en Java indépendante (il faut juste un *runtime* Java) et une *servlet* qui peut donc être exécutée à l'intérieur d'un navigateur Web bien né. Malheureusement, les dernières versions de cet excellent produit viennent de devenir commerciales.

Implémentation OpenSSH

Cette implémentation est aujourd'hui sans doute celle qu'il faut utiliser en environnement Unix/Linux (suivi constant, bonne réactivité, open-source et gratuite), mais n'est malheureusement pas exempte de défaut (à mes yeux).

Au départ, le développement de OpenSSH a débuté sur une base originale SSH 1.2.12 de Tatu Ylonen (la dernière à être suffisamment libre) dans le cadre de OpenBSD 2.6 (via OSSH). Actuellement, OpenSSH est développé par deux équipes, l'une n'effectuant que le développement relatif à OpenBSD, et l'autre adaptant constamment ce code pour en faire une version portable.

Tout cela a certaines conséquences, en particulier, le code est devenu et tend à être de plus en plus une monstrueuse adaptation constante (je sens le syndrome "*sendmail*" poindre à l'horizon) et cela n'est pas très sain pour une application de cryptologie qui devrait être extrêmement rigoureuse et claire (j'éviterai ici mon classique couplet sur les produits de cryptologie open-source "*que tout le monde peut relire mais que personne n'a jamais relu*", et qui alors dans les faits ne procurent qu'une confiance non démontrée et donc totalement artificielle, voire surfaite et surestimée).

En dehors de la propreté et de la lisibilité du code, deux autres points fondamentaux me perturbent :

- OpenSSH utilise pour les services de cryptographie la bibliothèque OpenSSL, et en général (soit que l'on utilise une distribution binaire packagée, soit que l'on construise le produit avec les

habitudes classiques), cette librairie est utilisée dynamiquement. Dans le cadre d'une implémentation d'un outil de cryptologie destiné à procurer un niveau de sécurité et confiance maximum, cette approche me semble totalement erronée. En effet, une attaque sur la bibliothèque vaudra attaque sur le produit. Et au-delà d'une attaque perverse, les caractéristiques (qualités) cryptographiques de OpenSSH sont/seront celles de la bibliothèque, qui vivra sa vie indépendamment de OpenSSH.
- OpenSSH utilise pour certains services sensibles (générateur de pseudo-aléas par exemple) les services système de OpenBSD. Dans ce seul contexte, je ferai déjà la même remarque de dépendance externe que celle concernant OpenSSL. Plus gênant encore, la version portable de OpenSSH, destinée à tourner sur d'autres plates-formes, délègue les services requis à OpenBSD à des mécanismes divers selon la plate-forme cible. Par exemple, selon la disponibilité ou non d'un générateur d'aléas sur le système, on l'utilisera, ou on en utilisera un interne pas vraiment validé. En conséquence, l'entropie effective de OpenSSH devient dépendante de la plate-forme d'exécution, voire moyennement déterministe.

Je pense pour ma part (et je ne suis pas le seul) qu'un produit de cryptologie multiplateforme devrait avoir un comportement démontrable (démontré) déterminé (et constant) quelle que soit la plate-forme, aussi bien en considérant (éliminant) les caractéristiques propres de celle-ci que son évolution dans le temps (releases système par exemple).

Ces réserves faites, les implémentations concurrentes libres n'étant pas si nombreuses et mieux loties, je crois qu'il est pragmatique de considérer qu'actuellement OpenSSH est la pire des implémentations, à l'exclusion de toutes les autres... ! Un chantier utile à la communauté serait une remise à plat et une reprise du codage à zéro cependant.

Mauvaises nouvelles...

SSH n'est pas miraculeux ! Il fait bien ce pour quoi il a été conçu, il ne faut pas en attendre plus. En particulier, il n'empêchera pas les connexions "autorisées" : si un compte est compromis par ailleurs, il sera bien entendu possible pour un intrus de se connecter via SSH sur votre machine, même si c'est le seul moyen, puisqu'il possède l'authentification. L'utilisation de SSH n'est donc pleinement efficace que corrélée à une politique et une pratique de la sécurité cohérente : si on possède le même mot de passe partout, et que l'on n'utilise pas SSH partout, le risque potentiel n'est que faiblement diminué. Il faut noter que dans cette situation, SSH peut même se "retourner" contre vous, car l'intrus utilisant un mode de connexion sécurisé (chiffré) permettant le tunneling, il pourra faire à peu près ce qu'il veut sans possibilité pour vous de le tracer efficacement.

Dans ce registre, il faut également noter que les "rootkits" bien faits contiennent en général un démon SSH pour permettre le retour discret chez vous, mais un peu modifié : il n'écoute pas sur le port standard 22 évidemment, il a le bon goût de ne plus rien loguer bien entendu, il est nommé comme un autre démon banal (*httpd* par exemple), et non visible sur un "*ps*" (qui lui est en général également un peu trafiqué par le même rootkit). Cela va sans doute venir, mais je n'ai pas encore vu de rootkit modifiant "*lsof*" qui devient alors un bon moyen de détecter des processus cachés par les rootkits classiques.

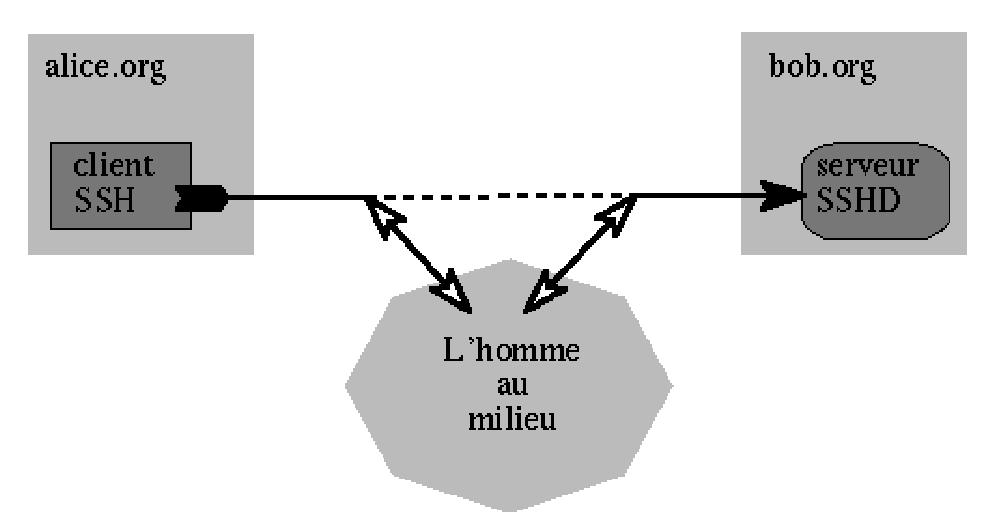
A contrario, il ne faut pas fantasmer trop sur le danger que représenterait un démon SSH permettant aux intrus de passer encore plus inaperçus : vous savez (j'espère) qu'il est possible de mettre n'importe quoi

dans quasi n'importe quoi en IP, y compris en "détournant" les protocoles indispensables même via un firewall : tunneling dans HTML, tunneling dans ICMP, tunneling dans DNS, ... Ce qui vous met en danger, c'est d'allumer votre ordinateur, pas d'y installer SSH !

L'utilisation de SSH doit donc naturellement être un élément cohérent d'une politique de sécurité globale, et en particulier doit se faire en complément d'une politique d'accès et de filtrage.

SSH n'est pas exempt non plus de "trous" de sécurité, qui peuvent se situer au niveau de l'implémentation (de nombreux ont été corrigés par le passé, il n'y a pas de programmes parfaits), mais aussi au niveau du protocole. Ces "trous", même s'il sont parfois annoncés de façon très alarmantes sont en général des faiblesses dont l'exploitation reste très technique et complexe à réaliser : il ne faut pas perdre de vue que les incidents de sécurité qui auraient pu être évités par la seule utilisation de SSH sont quotidiens, alors que ceux pour lesquels il aurait été démontré qu'ils ne sont dûs qu'à l'exploitation d'une faiblesse actuelle de SSH ne restent sans doute que théoriques. On lira avec intérêt cette étude concernant les attaques du type "*man in the middle*" : http://www.hsc.fr/ressources/presentations/mitm/index.html.fr. Il faudra cependant prendre en compte ce type de vulnérabilité potentielle dans le cadre d'applications de "haute-sécurité" (bancaire, militaire, ...), où les moyens de l'attaquant, très motivé par l'enjeu et le bénéfice escomptés, peuvent être importants.

- Attaque "man in the middle" :



L'attaquant intercepte les paquets des deux parties, et génère les siens afin de faire croire à chacun qu'il est l'autre (différents scénarios sont possibles, jusqu'à clôre la connexion d'un coté, et la poursuivre vers l'autre partie qui croit toujours avoir à faire au partenaire habituel)

J'aime aussi habituellement signaler une faiblesse incompréhensible du protocole (*) concernant le "padding" : aussi bien en protocole version 1 que 2, les paquets, pour avoir une longueur multiple de 64 bits, sont paddés avec un aléa. Cela est assez inhabituel, et ménage une faille classique et connue des implémentations de produits cryptologiques : un canal "caché" (ou "subliminal") (voir article sur la crypto dans ce même numéro pour définition et détails). Habituellement, on "padde" avec une séquence vérifiable comme par exemple inscrire la valeur *n* pour l'octet de rang *n* (*self describing padding*). Dans SSH, la séquence étant (par définition) aléatoire, elle n'est pas vérifiable. En conséquence, il est possible pour une des parties de la communication de l'utiliser pour pervertir / compromettre la communication, par exemple à l'usage d'un tiers à l'écoute. On peut également imaginer une implémentation pervertie à

l’insu des deux parties (facile à réaliser sur un produit fourni en binaire seul comme le sont en général les produits commerciaux). L’utilisation de ce canal est facile (*voir encadré*), et ne nécessite ici de pervertir que le client ou le serveur. Laisser cet énorme défaut dans le protocole, alors qu’il est universellement connu que l’installation d’un canal caché dans un produit de cryptologie est LE moyen classique et basique de le pervertir me semble incompréhensible (d’autant qu’il est immédiat de le supprimer). Il peut être intéressant à ce propos de lire ce papier de Bruce Schneier concernant l’implémentation de tels dispositifs dans certains produits influencés par des agences gouvernementales (http://www.counterpane.com/crypto-gram-9902.html#backdoors).

Je terminerai pour signaler que j’avais pour ma part, lors du portage de SSF, découvert un trou de sécurité dans le codage des versions Unix de SSH antérieures à la version 1.2.25 ayant pour conséquence que le générateur d’aléas était... prévisible... (cette situation pouvant être considérée comme fâcheuse dans un produit cryptographique, je n’entre pas dans les détails techniques, mais cela devait permettre de compromettre une communication en étant simplement à l’écoute). L’équipe de développement de SSH avait à l’époque bien entendu corrigé le problème (une seule ligne de code a corriger), mais curieusement, c’est un "bug" qui n’a fait l’objet d’aucune alerte de leur part (ni aucune autre part), ni même la moindre mention dans le "changelog" du produit... on aurait voulu que cela ne se sache pas qu’on aurait pas fait autrement. Aucun rapport bien entendu avec l’article sus-cité.

(*) Ce problème est clairement identifié, puisque le document de l’IETF "*SSH Transport Layer Protocol*" indique : "*The protocol was not designed to eliminate covert channels. For example, the padding, SSH_MSG_IGNORE messages, and several other places in the protocol can be used to pass covert information, and the recipient has no reliable way to verify whether such information is being sent.*". Je considère le canal caché dans le padding plus "grave" que les autres, car encore plus "caché"

Conclusion

Je reprendrai ce que j’ai écrit en introduction : SSH, sans être miraculeux ni résoudre tous les problèmes de sécurité à lui seul, permet de traiter tout à fait efficacement la majorité des aspects fragiles des modes de connexions interactifs historiques à moindre effort (quasi-nul). En conséquence, son utilisation devrait se substituer naturellement à celle des *telnet*, *rsh*, *rlogin* et autres *R-commandes* et modes de connexion interactifs. Tout incident de sécurité qui aurait été limité par l’usage de SSH est impardonnable !

Bibliographies, liens essentiels

Les deux ouvrages suivants couvrent SSH version 1 et SSH version 2 :

- *SSH : the Secure Shell*
  Daniel J. Barret & Richard E. Silverman
  O’Reilly - ISBN 0-596-00011-1
- *Unix Secure Shell*
  Anne Carasik
  McGraw-Hill - ISBN 0-07-134933-2

S’il faut ne retenir qu’un URL, celui-ci vous permettra de tout retrouver (en particulier la rubrique "Alternatives" pour les autres implémentations)

- http://www.openssh.com

Et si vous voulez à tout prix dépénser de l’argent, voici un départ tout indiqué... :

- http://www.ssh.com

Exploitation du canal caché induit par le padding aléatoire dans SSHv1

A la demande générale, merci de ne pas en faire mauvais usage... Voici donc comment exploiter le canal caché (subliminal) induit par le fait que le padding est aléatoire dans SSHv1 (et v2). Je décline toute responsabilité concernant les attaques cardiaques des plus paranoïaques ;-)

La structure d’un paquet SSHv1 est la suivante :

offset **(en octets)**	**nom**	**longueur** **(en octets)**	**usage**
0	taille	4	taille du paquet, champs taille et padding exclus, donc : taille = longueur(type)+longueur(data)+longueur(CRC)
4	padding	p =1 à 8	bourrage **aléatoire** : sa taille est ajustée pour que la partie chiffrée ait une longueur multiple de huit
4+p	type	1	type du paquet
5+p	données	n (variable >= 0)	
5+p+n	checksum	4	CRC32

Seul le champ "taille" n’est pas chiffré. La longueur de la partie chiffrée est toujours multiple de huit, ajustée grâce au "padding". Le padding est toujours présent, si la longueur des trois derniers champs est déjà multiple de huit, le padding comporte alors huit octets également ($5+p+n$ congrus à 0 modulo 8). Soit C la fonction de chiffrement, symétrique, effectuée en mode CBC, et C^{-1} celle de déchiffrement. Pour simplifier, on ne va traiter que les paquets dont le padding a exactement huit octets (un huitième statistiquement). Lorsqu’un tel paquet se présente, au lieu de le remplir avec un vrai aléa, on va y mettre un contenu $C^{-1}(M)$, de huit octets en l’occurrence, c’est à dire le déchiffrement d’un message M avec la fonction C utilisée pour chiffrer le canal (le fait que M soit "déchiffré" sans avoir été préalablement chiffré est sans importance d’un strict point de vue mathématique, je ne détaille pas les détails d’implémentation pratiques, comme la nécessité de dupliquer les contextes des routines de chiffrement

au bon moment pour cause de mode CBC). Ensuite, on effectue le traitement habituel du paquet, à savoir le chiffrement de celui-ci par tranches de huit octets.

Le résultat sera le suivant :

offset	contenu	note
0	taille	4 octets non chiffrés
4	8 octets de padding chiffrés	donc $C(C^{-1}(M))$
12... fin	type, data, CRC chiffrés	

Que se passe-t-il devant nos petits yeux ébahis ? Le premier bloc chiffré contient $C(C^{-1}(M))$. Hors, comme C est une fonction de chiffrement symétrique, $C(C^{-1}(M)) = M$. Ce premier bloc se trouve donc être émis en clair dans un flux chiffré ! Cela veut tout simplement dire que toute personne espionnant la communication et au courant du stratagème saura exploiter cette information. Bien entendu, il est tout à fait concevable que le message M soit lui-même chiffré par ailleurs (contre une clé publique par exemple, ce qui évite de déposer un secret dans le code perverti), ce qui le rend toujours indéchiffrable pour qui n'est pas au parfum.

Il suffit (par exemple) de trois paquets de ce type pour y passer la clé de session triple-DES (168bits) , et ensuite, l'observateur du flux pourra déchiffrer toute la communication. A partir du moment où la clé est transmise, il n'y a plus nécessité de "pre-déchiffrer" le padding perverti avant de l'injecter dans le paquet, il est possible d'utiliser les padding de tailles quelconques si on veut y ajouter encore plus de choses (des secrets prélevés sur le poste de travail par exemple, ce qui permet, en plus de révéler la communication, de faire "fuir" des secrets qui ne seraient présents que sur le poste sans avoir besoin de s'y connecter).

Les deux parties de la communication connaissant la ou les clés, il suffit qu'une seule des deux soit perverse pour que ce système fonctionne, à l'insu du plein gré de l'autre. On peut également imaginer qu'une des deux parties utilise un produit perverti à son insu (un binaire commercial par exemple, qui vérifie ?) au bénéfice d'un tiers, et ce tiers pourra compromettre la communication au détriment des deux parties.

L'utilisation de ce canal caché est *absolument indétectable* ! (il faut veiller cependant à surchiffrer comme indiqué le message caché afin que l'entropie du bloc ne révèle pas le stratagème). Il est indétectable, car comme par définition le padding est aléatoire, on ne peut vérifier aucune cohérence. Il ne faut *jamais* faire du padding aléatoire dans un produit de cryptologie, jamais.

Ce qui rend ce canal plus dangereux que les autres dans le protocole (celui induit par les messages de type SSH_MSG_IGNORE par exemple) est que celui-ci est utilisable *sans* connaitre préalablement les clés de chiffrement.

Pour éviter cet effet pervers du padding aléatoire, il suffit de convenir dans la définition du protocole que le padding est déterministe : couramment, on utilise du "*self describing padding*", c'est à dire que l'octet d'offset n contient n. Le padding est resté aléatoire dans SSH v2, c'est un choix, dont acte... (le format des paquets est différent, la méthode exposée ici doit donc être adaptée, sans plus).

Pour conclure, je dirais juste que si je critique ce canal caché, c'est parce que je souhaiterais qu'un produit comme SSH, qui se revendique de haute sécurité, offre véritablement toutes les garanties. Il n'échappera maintenant à personne qu'il doit être possible d'imaginer beaucoup de perversions potentielles dans les produits commerciaux actuels de toute nature : seuls les produits open-source offrent le prérequis indispensable, la possibilité de vérifier le code utilisé (même si cette vérification reste souvent à faire).

A-t-on le "droit" d'utiliser SSH en France ?

Question récurrente... A la date de rédaction de cet article, le régime légal français concernant la cryptologie (*) est toujours restrictif. Ce régime est celui-ci instauré par les (derniers) décrets de mars 1999, fixant en particulier une limite entre les régimes déclaratifs et d'autorisation préalable à 128 bits de clé. L'algorithme par défaut utilisé par SSH (en l'absence d'accord négocié sur un autre) est le "*triple-DES à trois clés*", légalement assimilé en France à un algorithme utilisant une clé de 168 bits. Formellement, il faudrait donc demander une autorisation préalable pour utiliser SSH à ce jour... Le Premier ministre a annoncé en janvier 1999 l'intention du gouvernement de totalement libéraliser l'usage de la cryptologie, mais les lois et décrets correspondants n'ont toujours pas vu le jour. La libéralisation de la cryptologie est incluse dans la "Loi sur la Société de l'Information" (au milieu de plein de choses sans aucun rapport...). Le projet de texte de cette LSI vient (juin 2001) d'être accepté en conseil des ministres, il ne reste plus qu'à le faire voter par le Parlement, et en publier les décrets d'application (cela peut prendre encore très longtemps...).

Et dans les faits ? Dans la vraie vie du vrai monde, personne ne sait vraiment comment s'appliquent les textes actuels à un produit comme OpenSSH, pour lequel il n'y a pas de "fournisseur" au sens des textes actuels, et qui formellement ne contient pas de code cryptographique... En effet, OpenSSH utilise (en général dynamiquement) la bibliothèque OpenSSL pour les services de chiffrement. OpenSSL est très largement importée, installée et utilisée en France pour en particulier implémenter des serveurs Web sécurisés (*https*). Et cela sans que pas grand monde (personne ?) n'ait déclaré quoi que ce soit, ni que bien grand monde ne s'en inquiète...

Bref, je ne vais pas développer plus longtemps les imprécisions et contraintes du régime légal (actuel) français concernant la cryptologie (pendant ce temps, l'Etat allemand, plus pragmatique peut-être, finance le développement de GnuPG et vire Microsoft^tm des applications militaires...). Je ne retiendrai que cet extrait du projet de la LSI : "Article 37, alinéa 1 : *L'utilisation des moyens de cryptologie est libre*", qui résume, j'espère, le régime à venir. Pour une réponse officielle, l'avis du DCSSI doit être celui autorisé. A noter pour terminer, la recommandation du Parlement Européen d'utiliser le chiffrement (et les produits open-source) pour sécuriser les communications dans son rapport très intéressant concernant le réseau Echelon (http://www.europarl.eu.int/tempcom/echelon/pdf/prechelon_fr.pdf).

(*)(j'utilise dans ce paragraphe le terme *cryptologie* dans le sens ou c'est celui utilisé par les textes légaux, même si cela ne correspond pas nécessairement à la signification technique stricte)
